Where did we leave off?

```
lemma radius_pos (a b : Pt) (α : Circ)
    (a_cent_of_α : Center_of_Circ a α)
    (b_on_α : Pt_on_Circ b α) :
    0 < dist a b := by
```

Main idea: axiom

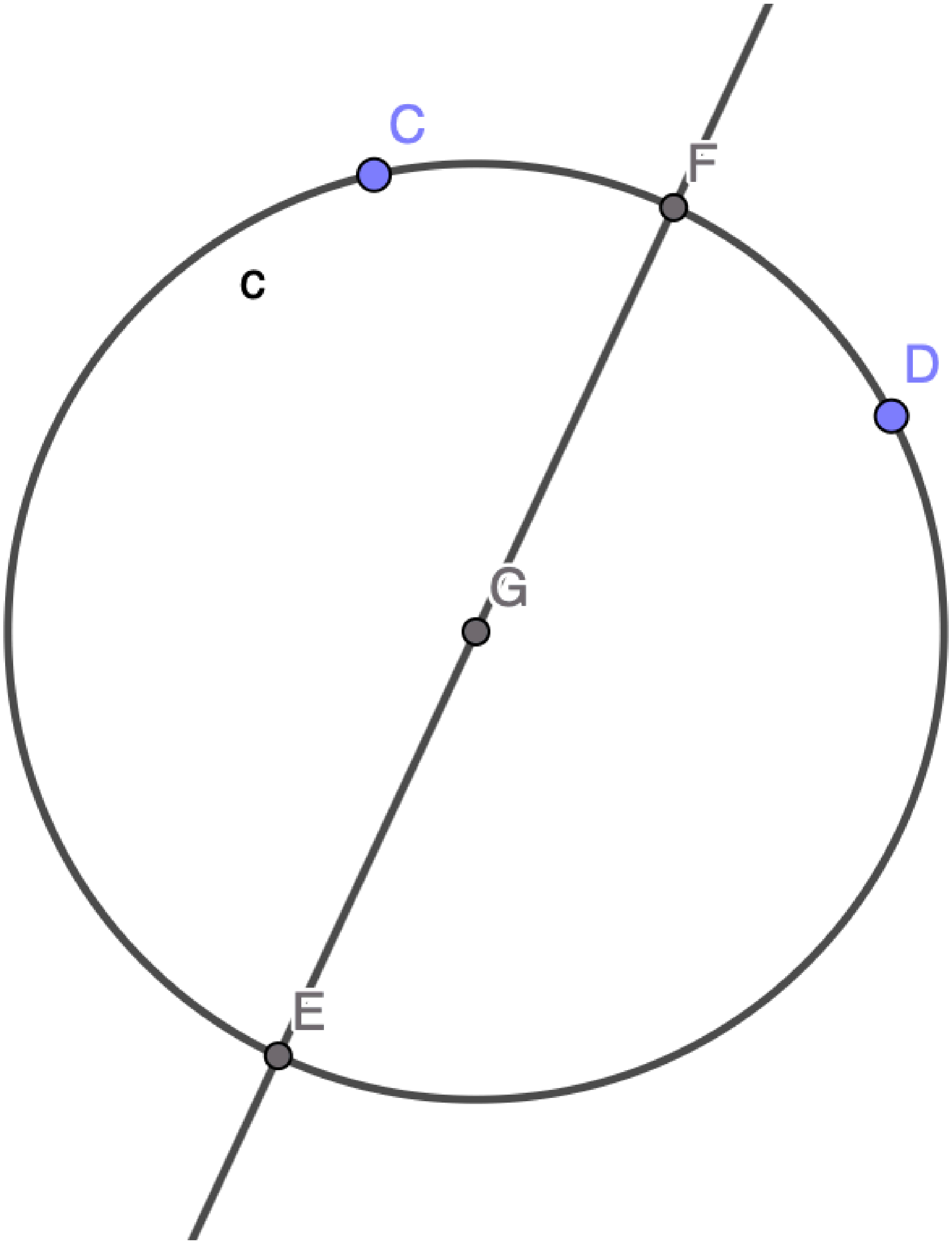
```
Pts_of_Circ : ∀ (α : Circ), ∃ (b c : Pt), Pt_on_Circ b α ∧ Pt_on_Circ c α ∧
b ≠ c
```

tells us that there are *two distinct* points on any circle! Which will somehow imply that a circle can't consist of a single point (i.e. have radius zero)
Question: Why do we even want to have two distinct points on a circle? Why not assume we only have one?

Challenge question: Given a circle with no center, how to find the center???

[Hint: start with two distinct points on the circle!]
Solution: Step 1: get two distinct points on the circle
Step 2: by straightedge and compass, draw the perpendicular bisector of those two points
Step 3: by some process (???) this line intersects the circle in two points, E and F

Step 4: the midpoint of E and F is the center



Back to original problem of positivity of radius:

By `dist_nonneg`, either `dist a b = 0` or `0 < dist a b` -- and in the latter case, we're done. Formally, first let's record:

```
have : 0 ≤ dist a b := dist_nonneg a b
```

And then we want to prove

```
have : 0 = dist a b ∨ 0 < dist a b := by
```

and this is a basic statement about the real numbers, not something about geometry, so the name for the theorem that closes this goal is not known to us. But it's such a basic fact that it must be somewhere in the Lean Mathlib library. The way to search for a theorem that you suspect is `exact`ly something already in the library is to write `exact?`. When you do that, if that theorem already exists, you'll see in the goal state something like:

```
Try this: exact Decidable.eq_or_lt_of_le this
```

And when you click on the exact, it replaces your text with the text that solves the goal.
Next step: We now have a hypothesis:

```
this : 0 = dist a b ∨ 0 < dist a b
```

We can't use `constructor` because (1) not in the goal, and (2) not `∧`. We want to break the logic by cases according to which case it is. That's called: `by_cases`.
More generally, if we have a hypothesis `H : P ∨ Q`. Then writing

```
cases this with
| inl dist_eq =>
    sorry
| inr dist_notEq => exact dist_notEq
```

splits the goal into two goals, one assuming that `P` is true, and another assuming that `Q` is true. In our case, `Q` is exactly the original goal, so `exact dist_notEq` solves everything. We need to see what happens in case `P`, where the game board says:

```
dist_eq : 0 = dist a b
⊢ 0 < dist a b
```

We're obviously in a state of contradiction. If we can prove `false`, then we can prove whatever we want (we must've stumbled into a contradiction). Note: we're trying to prove that the radius is always positive, so this is the part of the argument that says the radius can't be zero.
To replace any goal by `false`, the tactic is called `exfalso`.

...

If you have `H1 : X = Y` and `H2 : Y = Z`, then `have H3 : X = Z := H1.trans H2`

...

We've gotten to the point where we proved `e = f` and `e ≠ f`. Recall that the latter is by definition: `e = f → False`.

```
import Mathlib

class EuclideanPlane where
  Pt : Type
  Line : Type
  Circ : Type
  Pt_on_Line : Pt → Line → Prop
  Line_of_Pts : ∀ a b : Pt, ∃ L : Line, (Pt_on_Line a L) ∧ (Pt_on_Line b L)
  Pt_on_Circ : Pt → Circ → Prop
  Center_of_Circ : Pt → Circ → Prop
  Circ_of_Pts : ∀ a b : Pt, ∃ α : Circ, (Center_of_Circ a α) ∧ (Pt_on_Circ b
α)
  dist : Pt → Pt → ℝ
  dist_symm : ∀ (a b : Pt), dist a b = dist b a
  dist_nonneg : ∀ (a b : Pt), 0 ≤ dist a b
  dist_pos_def : ∀ (a b : Pt), dist a b = 0 ↔ a = b
  Circs_inter : Circ → Circ → Prop
  Pt_in_Circ : Pt → Circ → Prop
  Circs_inter_iff : ∀ (α β : Circ), Circs_inter α β ↔
    (∃ (a b : Pt), Pt_on_Circ a α ∧ Pt_in_Circ a β ∧
      Pt_on_Circ b β ∧ Pt_in_Circ b α ∧ a ≠ b)
  Pt_on_Circ_iff : ∀ (a b c : Pt) (α : Circ), Center_of_Circ a α →
    Pt_on_Circ b α → (Pt_on_Circ c α ↔ dist a c = dist a b)
  Pt_in_Circ_iff : ∀ (a b c : Pt) (α : Circ), Center_of_Circ a α →
    Pt_on_Circ b α → (Pt_in_Circ c α ↔ dist a c < dist a b)
  Pts_of_Circ : ∀ (α : Circ), ∃ (b c : Pt), Pt_on_Circ b α ∧ Pt_on_Circ c α
∧ b ≠ c

variable [EuclideanPlane]

namespace EuclideanPlane

def IsEquilateralTriangle (a b c : Pt) : Prop := (dist a b = dist b c) ∧
(dist a b = dist a c)

-- An easier helper lemma:
lemma dist_self (a : Pt) : dist a a = 0 := by
```

```
  have := (dist_pos_def a a).2
  apply this
  -- now the goal is to prove `⊢ a = a` and that's solved by
  -- the reflexive property of the equal sign, called `rfl`
  rfl

-- Another helper lemma: the "radius" is always strictly positive
lemma radius_pos (a b : Pt) (α : Circ)
    (a_cent_of_α : Center_of_Circ a α)
    (b_on_α : Pt_on_Circ b α) :
    0 < dist a b := by
  -- idea:
  -- by `dist_nonneg`, either `0 = dist a b` or `0 < dist a b`
  have : 0 ≤ dist a b := dist_nonneg a b
  have : 0 = dist a b ∨ 0 < dist a b := by
    exact Decidable.eq_or_lt_of_le this

  cases this with
  | inl dist_eq =>
    exfalso
  -- Now we're arguing by contradiction (we want to prove `false`!)
    have := Pts_of_Circ α
    obtain ⟨e, f, e_on_α, f_on_α, e_ne_f⟩ := this
  -- Using `Pts_of_Circ`, we can `obtain` two new points, say, `e` and
  -- `f` that are distinct and both on `α`.

    have a_eq_b : a = b := (dist_pos_def a b).1 dist_eq.symm
  -- If `dist a b = 0` then `a = b` (by `dist_pos_def`).
  -- But `Pt_on_Circ_iff` says that radii are well-defined
  -- that is, `dist a e = dist a b = 0 = dist b e` So
    have := (Pt_on_Circ_iff a b e α a_cent_of_α b_on_α).1 e_on_α
    have := this.trans dist_eq.symm
    have a_eq_e : a = e := (dist_pos_def a e).1 this

    have := (Pt_on_Circ_iff a b f α a_cent_of_α b_on_α).1 f_on_α
    have : dist a f = 0 := this.trans dist_eq.symm
    have a_eq_f : a = f := (dist_pos_def a f).1 this

    have e_eq_f : e = f := a_eq_e.symm.trans a_eq_f
  -- `b = e` (again by `dist_pos_def`). The same argument proves
```

```
  -- `b = f`, which by transitivity proves `e = f`.
  -- Which is a contradiction!
  -- Hint: `e ≠ f` by defintion means: `e = f → false`.

    exact e_ne_f e_eq_f
  | inr dist_notEq => exact dist_notEq
  -- if the second case, then we're done. So we just want to rule
  -- out the first case

-- Let's make a helper lemma: given a circle `α` and a point `a` that's the
center of the circle, `a` is in the interior of the circle
lemma in_Circ_of_center (a : Pt) (α : Circ) (a_cent_α : Center_of_Circ a α)
:
    Pt_in_Circ a α := by
  have := Pts_of_Circ α
  obtain ⟨b, c, b_on_α, c_on_α, b_ne_c⟩ := this

  have := (Pt_in_Circ_iff a b a α a_cent_α b_on_α).2
  apply this

  have dist_aa : dist a a = 0 := dist_self a

  sorry -- HOMEWORK

theorem PropIp1 (a b : Pt) : ∃ (c : Pt), IsEquilateralTriangle a b c := by
  have Step1 : ∃ (α : Circ), Center_of_Circ a α ∧ Pt_on_Circ b α :=
    Circ_of_Pts a b
  obtain ⟨α, a_center_of_α, b_on_α⟩ := Step1
  have Step2 : ∃ (β : Circ), Center_of_Circ b β ∧ Pt_on_Circ a β :=
    Circ_of_Pts b a
  obtain ⟨β, b_center_of_β, a_on_β⟩ := Step2
  have Step3 : Circs_inter α β := by
    have := (Circs_inter_iff α β).2
    apply this
    use b, a
    constructor
    · exact b_on_α
    · constructor
      ·
        have := (Pt_in_Circ_iff b a b β b_center_of_β a_on_β).2
```

```
      sorry -- Pt_in_Circ b β
    · constructor
      · exact a_on_β
      · sorry -- Pt_in_Circ a α
have Step4 : ∃ (c d : Pt), Pt_on_Circ c α ∧ Pt_on_Circ d α ∧
  Pt_on_Circ c β ∧ Pt_on_Circ d β ∧ c ≠ d :=
    Pts_of_Circs_inter α β Step3
obtain ⟨c, d, c_on_α, d_on_α, c_on_β, d_on_β, c_ne_d⟩ := Step4

have Step5 : dist a c = dist a b :=
  (Pt_on_Circ_iff a b c α a_center_of_α b_on_α).1 c_on_α

have Step6 : dist b a = dist b c :=
  (Pt_on_Circ_iff b c a β b_center_of_β c_on_β).1 a_on_β

use c
unfold IsEquilateralTriangle
constructor
· convert Step6 using 1
  have := dist_symm a b
  exact this
· have Step7 := Step5.symm
  exact Step7
```

Homework: Prove `in_Circ_of_center`